Design Lab 14 *6.01 – Fall 2011*

I'm the Map!

Goals: In this lab, you will build a system that can run on the robot, allowing it to dynamically build a map of the obstacles around it as the robot moves through the world, progressing along a dynamically planned path to its goal. This system will operate robustly, despite unreliable sensors and uncertain obstacles. You will:

- Construct a dynamic map maker which uses perfect sonar information, and test it in with the planner of **Software Lab 14**, in Soar
- Improve your map maker to process imperfect sonar information
- Build a robust map maker using Bayesian state estimation
- Demonstrate a complete mapping and planning system with a real robot
- Optimize your system, eg by improving plans or adding heuristic rules to the search algorithms, and compete with your robot in a race!

- **Checkoffs**: The checkoffs are due during design lab this week, and software lab next week. You are expected to work during both labs, but are not expected to do any work outside of lab time.
- **Windows**: The graphics software that we are using sometimes crashes under Windows Vista and Windows 7. Please use a lab laptop instead.

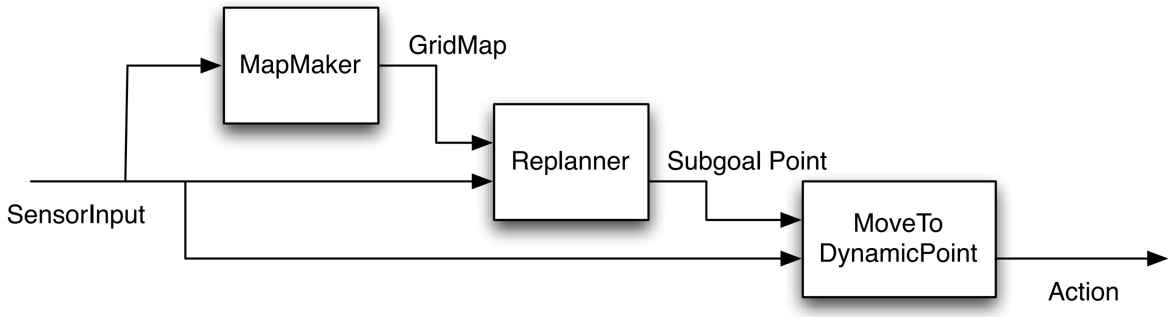
Resources: Do the lab on a computer that can run `soar`. These are the relevant files:

- `mapMakerSkeleton.py`: file to write your map maker code in
- `mapAndReplanBrain.py`: brain file to run the map maker
- `bayesMapSkeleton.py`: file to write your Bayesian map representation in
- `robotRaceBrain.py`: brain file for running on the real robot
- `mapAndRaceBrain.py`: brain file for running in simulation; prints out timing information
- `dl14World.py`, `bigPlanWorld.py`, `raceWorld.py`, `lizWorld.py`, `mazeWorld.py`, `mapTestWorld.py`: Soar world files

1 Introduction

In this lab, we will connect the planner from **Software Lab 14** with a state machine (named `MapMaker`) that dynamically builds a map as the robot moves through the world. The robot will, optimistically, start out by assuming that all of the locations it does not know about are free of obstacles, it will make a plan on that basis, and then begin executing the plan. But, as it moves, it will see obstacles with its sonars, and add them to its map. If it comes to believe that its current plan is no longer achievable, it will plan again. Thus, starting with no knowledge of the environment, the robot will be able to build a map. We'll start by building a simple map maker, then see what happens as the sensor data becomes less reliable, then adapt the map maker to handle unreliable sensor data.

Here is a diagram of the architecture of the system we will build.



Our architecture has three modules. We will give you our implementations of the replanner and the module that moves to a given point; you will concentrate on the mapmaker.

The `move.MoveToDynamicPoint` class of state machines takes instances of `util.Point` as input, and generates instances of `io.Action` as output. That means that the point that the robot is 'aiming' at can be changed dynamically over time. (Remember that you wrote a machine like this in Design Lab 3!).

The `replanner.ReplannerWithDynamicMap` state machine takes a `goalPoint` as a parameter at initialization time. The `goalPoint` is a `util.Point`, specifying a goal for the robot's location in the world, which will remain fixed. The robot's sensor input (which contains information about the robot's current location) as well as the `DynamicGridMap` instance that is output by the `MapMaker` will be the inputs to this machine. The replanner makes a new plan on the first step, draws it into the map, and outputs the first 'subgoal' (that is, the center of the grid square that the robot is supposed to move to next), which is input to the driving state machine. On subsequent steps the replanner does two things:

1. It checks to see if the first or second subgoal locations on the current plan are blocked in the world map. If so, it calls the planner to make a new plan.
2. It checks to see if it has reached its current subgoal; if so, it removes that subgoal from the front of its stored plan and starts generating the next subgoal in the list as output.

2 Mapmaker, mapmaker, make me a map

Your job is to write a state-machine class, `MapMaker`, in the file `mapMakerSkeleton.py`. It will take as input an instance of `io.SensorInput`. Its state will be the map we are creating, which can be represented using an instance of `dynamicGridMap.DynamicGridMap`, which is like `basicGridMap.BasicGridMap`, but instead of creating the map from a file, it allows the map to be constructed dynamically. The grid map will be both the state and the output of this machine. The starting state of the mapmaker can just be the initial `dynamicGridMap.DynamicGridMap` instance.

For efficiency reasons, we are going to violate our state machine protocol and say that your `getNextValues` method should return *the same instance* of `dynamicGridMap.DynamicGridMap` that was passed in as the old state, as the next state and the output. It should make changes to that map using the `setCell` and `clearCell` methods. (We don't need `clearCell` in this section of the lab, but we will use it later). If we were to copy it every time, the program would be painfully slow.

The `dynamicGridMap.DynamicGridMap` class provides these methods:

- `__init__(self, xMin, xMax, yMin, yMax, gridSquareSize)`: initializes a grid with minimum and maximum real-world coordinate ranges as specified by the parameters, and with grid square size as specified. The grid is stored in the attribute `grid`. Initially, all values are set to `False`, indicating that they are **not** occupied.
- `setCell(self, (ix, iy))`: sets the grid cell with indices `(ix, iy)` to be occupied (that is, sets the value stored in the cell to be `True`).
- `clearCell(self, (ix, iy))`: sets the grid cell with indices `(ix, iy)` to be **not** occupied (that is, sets the value stored in the cell to be `False`).
- `occupied(self, (ix, iy))`: returns `True` if the cell with indices `(ix, iy)` is occupied by an obstacle.
- `robotCanOccupy(self, (ix, iy))`: returns `True` if it is safe for the robot to have its center point anywhere in this cell.
- `squareColor(self, (ix, iy))`: returns the color that the grid cell at `(ix, iy)` should be drawn in; in this case, it draws a square in black if it is marked occupied by an obstacle. It draws squares in gray that are not occupied by obstacles but are not occupiable by the robot because they are too close to an obstacle square; the gray cells are computed by `robotCanOccupy`.

What should the mapmaker do? The most fundamental thing it knows about the world is that the grid cell at the very end of a sonar ray is occupied by an obstacle. So, on each step, for each sonar sensor, if its value is less than `sonarDist.sonarMax`, you should mark the grid cell containing the point at the end of the sonar ray in the map as containing an obstacle. The `sonarHit` procedure you wrote in tutor problem **Wk.11.1.6** is available as

```
sonarDist.sonarHit(dist, sonarPose, robotPose)
```

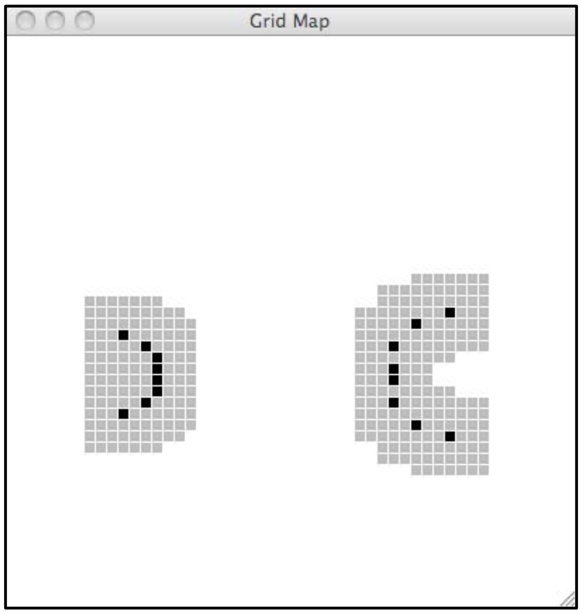
A list of the poses of all the sonar sensors, *relative to the robot*, is available in `sonarDist.sonarPoses`.

Step 1.

Check Yourself 1. We have defined a `SensorInput` class for testing in idle that simulates the `io.SensorInput` class in soar. Consider these two possible sensor input instances (each has a list of 8 real-valued sonar readings and a pose).

```
testData = [SensorInput([0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2],
                        util.Pose(1.0, 2.0, 0.0)),
            SensorInput([0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4],
                        util.Pose(4.0, 2.0, -math.pi))]
```

Be sure you understand why they give rise to the map shown below. Remember that the black squares are the only ones that are marked as occupied as a result of the sonar readings; the gray squares are the places that the robot cannot occupy (because it would collide with one of the black locations).



Step 2. Implement the `MapMaker` class. It will be called as follows:

```
MapMaker(xMin, xMax, yMin, yMax, gridSquareSize)
```

Remember to initialize the `startState` attribute and to define a `getNextValues` method that marks the cells at the end of the sonars rays in the input.

Step 3. Test your map maker inside idle (be sure to start with the `-n` flag). by doing this:

```
testMapMaker(testData)
```

It will make an instance of your `MapMaker` class, and call `transduce` on it with the `testData` from the Check Yourself question. Verify that your results match those in the figure.

Step 4. Now, test your code in soar. The file `mapAndReplanBrain.py` contains the necessary state machine combinations to connect all the parts of the system together into a brain that can run in soar.

You can work in any of the worlds described in the top of the brain file; select the appropriate simulated world in soar, and then be sure that you have a line like

```
useWorld(dl14World)
```

with a name corresponding to the world you are using, that selects the appropriate dimensions for the world you're working in (the line selecting `dl14World` is currently in the brain). Be sure to use the simulated world corresponding to the world file you have selected when you test your code.

A window will pop up that shows the current state of the map and plan. Black squares are those the map maker has marked as occupied. Sometimes squares will be drawn in gray: that means that, although they are not occupied by obstacles, they are not occupiable by the robot. Not all such non-occupiable squares will be drawn in gray (we don't want to redraw the whole screen too often), however.

Checkoff 1. **Wk.14.2.1**: Show the map that your mapmaker builds to a staff member. If it does anything surprising, explain why. How does the dynamically updated map interact with the planning and replanning process?

3 A noisy noise annoys an oyster

Step 5. By default, the sonar readings in soar are perfect. But the sonar readings in a real robot are nothing like perfect. Find the line in `mapAndReplanBrain.py` that says:

```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: noNoise
```

and change it to one of

```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: smallNoise
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: mediumNoise
```

This increases the default variance (width of gaussian distribution) for the sonar noise model to a non-zero value.

Check Yourself 2. Run the brain again in these noisier worlds. Why doesn't it work? How does the noise in the sensor readings affect its performance?

In fact, we get more information from the sonar sensors than just the fact that end of the ray is occupied. We also know that the grid cells along the sonar ray, between the sensor and the very last cell, are clear. Even when the sonar reading is greater than the maximum good value, you might consider marking the cells along the first part of the ray clear.

Step 6. Improve your `MapMaker` class to take advantage of this information. You will probably find the procedure `util.lineIndices(start, end)` useful: `start` and `end` should each be a pair of (x, y) integer grid cell indices; the return value is a list of (x, y) integer grid cell index pairs that constitute the line segment between `start` and `end` (including both `start` and `end`). You can think of these cells as the set of grid locations that could reasonably be marked as being clear, based on a sonar measurement. *Be sure not to clear the very last point, which is the one that you are already marking as occupied; although it might work now, if you clear and then mark that cell each time, it will cause problems in Section 4, when we use a state estimator to aggregate the evidence we get about each grid cell over time.*

Step 7. Test your new `MapMaker` in Idle by doing

```
testMapMakerClear(testClearData)
```

Note that this is `testMapMakerClear`, which is a different procedure from `testMapMaker`. It will create an instance of your map maker and set all of the grid squares to be occupied, initially. Then it will call `transduce` with this input:

```
testClearData = [SensorInput([1.0, 5.0, 5.0, 1.0, 1.0, 5.0, 5.0, 1.0],
                             util.Pose(1.0, 2.0, 0.0)),
                 SensorInput([1.0, 5.0, 5.0, 1.0, 1.0, 5.0, 5.0, 1.0],
                             util.Pose(4.0, 2.0, -math.pi))]
```

Check Yourself 3. Predict what the resulting map should look like, and make sure your code produces the right thing.

Step 8. Run `mapAndReplanBrain` in soar again and make sure you understand what happens with both no noise and medium noise.

Checkoff 2. **Wk.14.2.2**: Show your new map maker running, first with no noise and then with medium noise. We don't necessarily expect it to work reliably: but you should explain what it's doing and why.

4 Bayes Map

Checkoffs 3 and 4 are due in Software Lab 15, but we strongly encourage you to stay in Design Lab 14 and try to finish it. If you get checked off during the Design Lab, then you do not need to attend Software Lab 15.

One way to make the mapping more reliable in the presence of noise is to treat the problem as one of *state estimation*: we have unreliable observations of the underlying state of the grid squares, and we can aggregate that information over time.

Our space of possible hypotheses for state estimation should be the space of all possible maps. But if our map is a 20 by 20 grid, then the number of possible map-grids is 2^{400} (each cell can either be occupied or not, and so this is like the number of 400-digit binary numbers), which is *much* too large a space to do estimation in. In order to make the problem computationally tractable, we will make a very strong **independence assumption: the state of each square of the map is independent of the states of the other squares.** If we do this, then, instead of having one state estimation problem with 2^{400} states, we have 400 state estimation problems, each of which has 2 states (the grid cell can either be occupied or not).

Luckily, we have already built a nice state-machine class for state estimation, and we can use it to build a new subclass of `dynamicGridMap.DynamicGridMap`, where each cell in the grid contains **an instance of `seFast.StateEstimator`** (which you implemented in **Wk.11.1.3**).

Your job is to write the definition for the `BayesGridMap` class, in the file `bayesMapSkeleton.py`. Before doing this, you'll need to think through how to use state estimators as elements of the grid.

Recall that the argument to the `__init__` method of `seFast.StateEstimator` is **an instance of `ssm.StochasticSM`**, which specifies the dynamics of the environment. Here are some points to think about when specifying the world dynamics of a single map grid cell:

- There are two possible states of the cell: occupied or not.
- There are two possible observations we may make of this cell: it is free, or it was the location of a sonar hit.
- You can assume that the environment is completely static: that is, that the actual state of a grid cell never changes, even though your belief about it changes as you gather observations. But, if you want to, you can also consider the situation where the environment changes, perhaps because furniture is moved.

Check Yourself 4. Remember that the sonar beams can sometimes bounce off of obstacles and not return to the sensor, and that when we say a square is clear, we say that it has nothing anywhere in it. What do you think the likelihood is that we observe a cell to be free when it is really occupied? That we observe it as a hit when it is really not occupied? What should the prior (starting) probabilities be that any particular cell is occupied?
Decide on possible values for the state of the cell. Assume that the observation can be either `'hit'`, if there is a sonar hit in the cell or `'free'` if the sonar passes through the cell. **To forestall confusion, pick names for the internal states that are neither `'hit'` nor `'free'`.**
If you are having trouble formulating the starting distribution, observation and transition models for the state estimator, talk to a staff member.

Step 9. Write code in `bayesMapSkeleton.py` to create an instance of `ssm.StochasticSM` that models the behavior **of a single grid cell.**

Step 10. Test your grid cell model by doing

```
testCellDynamics(cellSSM, yourTestInput)
```

where `cellSSM` is an instance of `ssm.StochasticSM` and `yourTestInput` is one of the lists below. It will create an instance of a state estimator for a single grid cell and feed it a stream of observations. Then it will call `transduce` with the data input.

What is its final degree of belief that the cell is occupied if you give it this input data? (Why are the Nones here?)

```
mostlyHits = [('hit', None), ('hit', None), ('hit', None), ('free', None)]
```

How about if you give it this input data?

```
mostlyFree = [('free', None), ('free', None), ('free', None), ('hit', None)]
```

Step 11. Now it is time to **think** through a strategy for implementing the BayesGridMap class; don't start implementing just yet

You will have to manage the initialization and state update of the state estimator machines in each cell yourself. You should be sure to call the start method on each of the state-estimator state machines just after you create this grid. You will also, whenever you get evidence about the state of a cell, have to call the step method of the estimator, with the input (o, a), where o is an observation and a is an action; we will be, effectively, ignoring the action parameter in this model, so you can simply pass in None for a.

You can remind yourself of the appropriate methods for creating a state estimator and for starting and stepping a state machine by looking at the online software documentation. Once a state machine mysm has been started, you can access its internal state with mysm.state.

Your BayesGridMap will be a subclass of DynamicGridMap and can be modeled directly on the following aspects of DynamicGridMap.py:

```
class DynamicGridMap(gridMap.GridMap):
    def makeStartingGrid(self):
        return util.make2DArray(self.xN, self.yN, False)
    def squareColor(self, (xIndex, yIndex)):
        if self.occupied((xIndex, yIndex)): return 'black'
        else: return 'white'
    def setCell(self, (xIndex, yIndex)):
        self.grid[xIndex][yIndex] = True
        self.drawSquare((xIndex, yIndex))
    def clearCell(self, (xIndex, yIndex)):
        self.grid[xIndex][yIndex] = False
        self.drawSquare((xIndex, yIndex))
    def occupied(self, (xIndex, yIndex)):
        return self.grid[xIndex][yIndex]
```

Step 12.

Wk.14.2.3 Solve this tutor problem on making collections of object instances.

Step 13. Here is some further description of the methods you'll need to write. Remember that the grid of values in a DynamicGridMap is stored in the attribute grid. We don't need to write the __init__ method, because it will be inherited from DynamicGridMap.

- makeStartingGrid(self): Construct and return two-dimensional array (list of lists) of instances of seFast.StateEstimator. You can use the attributes xN and yN of self to know how big to make the array. You should use util.make2DArrayFill for this (be sure you understand why make2DArray is not appropriate).

- `setCell(self, (xIndex, yIndex))`: This method should do a state-machine update on the state machine in this cell, for the observation that there is a sonar hit in this cell. And it should redraw the square in the map in case its color has changed.
- `clearCell(self, (xIndex, yIndex))`: This method should do a state-machine update on the state machine in this cell, for the observation that this cell is free. And it should redraw the square in the map in case its color has changed.
- `occProb(self, (xIndex, yIndex))`: This method returns a floating point number between 0 and 1, representing the probability with which we believe that the specified cell is occupied. This is used for display purposes by the `squareColor` method, which has already been written.
- `occupied(self, (xIndex, yIndex))`: This method returns `True` if the cell should be considered to be occupied for the purposes of planning and `False` if not. You may have to experiment with this a bit in order to find a good threshold on the probability that the square is occupied. Use the `occProb` method specified above.

Step 14. Now, implement the `BayesGridMap` class in `bayesMapSkeleton.py`. It already has the `squareColor` method defined.

Step 15. Test your code in Idle by:

- Changing your `MapMaker` to use `bayesMap.BayesGridMap` instead of `dynamicGridMap.DynamicGridMap`. No further change to that class should be necessary.
- Running `mapMakerSkeleton.py` in Idle, and then typing in the shell:

```
testMapMakerN(1, testData)
```

It will do an update with the same data as we used in with the dynamic grid map. Now, the window that pops up uses a different color scheme: white means likely to be clear and bright green means likely to be blocked, with continuous variation between the colors. If a cell is considered to be blocked, it is colored black; if a cell is not blocked, but is also not occupiable by the robot, it is colored red.

If you type

```
testMapMakerN(2, testData)
```

then it will update the map 2 times with the given data.

Check Yourself 5. Try it with two updates. Try it with testClearData. Be sure it all makes sense.

Step 16. Now, test your mapper in soar, by running `mapAndReplanBrain` as before. You might find it particularly useful to use the step button.

Checkoff 3. **Wk.14.2.4**: Demonstrate your mapper in `mapAndReplanBrain` using your `BayesMap` module with medium noise and high noise. If it doesn't work with high noise, explain what the issues are, but you don't necessarily need to have it working with high noise.

4.1 Real robot

Now, let's see how well this works in the real world! Take your laptop to one of the real-world playpens, connect it to a robot, and run `robotRaceBrain.py`. You may have to adjust the parameters in your state estimator (typically, the false-positive rate, or the threshold for considering a square to be blocked) in order for it to work reliably.

Checkoff 4. **Wk.14.2.5**: Demonstrate your mapper on a real robot. You can move the obstacles around in the playpen for added fun, but be sure that you don't make it impossible to go from the start to the goal.

5 Optional: Go, speed racer, go!

In **Software Lab 14**, we worked on speeding up planning time by using **A* search with a heuristic**. And our robots can avoid obstacles by building a map and planning paths to avoid them. Now, we're going to work on making our robots move more quickly through the world. Your job during this lab is to speed up your robot as much as possible; at the end, we'll have a race.

In this section we will concentrate on the `Replanner` and `MoveToDynamicPoint` modules.

The `mapAndRaceBrain.py` is currently set up to work in `raceWorld.py`: select that as the simulated world, and run the brain. To change the world you're working in, change the `useWorld` line in the brain (and remember to change the simulated world, as well).

When you run using `mapAndRaceBrain.py`, you'll notice that when the robot reaches its goal, it stops and prints out something like

```
Total steps: 320
Elapsed time in seconds: 209.554840088
```

That's the number of soar primitive steps it took to execute your plan, and the amount of elapsed time it took. These numbers will be your 'score'. Note that we are aiming for low scores!

You can debug on your own laptop or an a lab laptop, but scores will only be considered official if they are run on a lab laptop.

Step 17.

Check Yourself 6. Run your robot through `raceWorld` and see what score you get. Write this down, because it's your baseline for improvement.

Note that there are several things that slow your robot down as it executes its plans:

- Each individual step, from grid cell to grid cell, is controlled by a proportional controller in `move.MoveToFixedPoint`. The controller has to slow down to carefully hit each subgoal.
- Rotations take a long time.

Below are some possible strategies for addressing these problems. You don't need to do any or all of these. If you pick one of your own (which we encourage!), talk to a staff member.

You can speed up the robot by producing a plan that requires less stopping and/or less turning. Implement these by editing your `GridDynamics` class (you can copy it from the file `swLab14/plannerStandaloneSkeleton.py` into `replannerRace.py`) or the `ReplannerWith-DynamicMap` class in `replannerRace.py` (read that code carefully).

1. Plan with the original set of actions, but then post-process the plan to make it more efficient. If the plan asks the robot to make several moves along a straight line, you can safely remove the intermediate subgoals, until the location where the robot finally has to turn.
2. Augment the space in which you are planning to include the robot's heading. Add an additional penalty for actions that cause the robot to rotate. Experiment with the penalty to improve your score. (This is pretty hard to get right; only do it if you have lots of spare time).
3. Increase the set of actions, to include moves that are more than one square away. You can use the procedure **`util.lineIndicesConservative((ix1, iy1), (ix2, iy2))`** to get a list of the grid cells that the robot would have to traverse if it starts at `(ix1, iy1)` and ends at `(ix2, iy2)`. This list of grid cells is conservative because it doesn't cut any corners.

You can also speed up the execution of the paths by changing the gains and tolerances in **the `move.MoveToDynamicPoint`** behavior (in `move.py`). Read the code in the file to understand what these parameters mean, and then consider adjusting them to improve the robot's behavior. But be sure you do not cause crashes into obstacles! You can edit these lines of code in `mapAndRace-Brain.py`.

```
move.MoveToFixedPoint.forwardGain = 1.0
move.MoveToFixedPoint.rotationGain = 1.0
move.MoveToFixedPoint.angleEps = 0.05
```

You are not allowed to change the `maxVel` parameter.

Step 18. Implement some improvements to make the robot go faster.

Check Yourself 7. Post your best scores in simulation on `raceWorld` and `lizWorld` on the board.

Step 19. Run on a real robot, using `robotRaceBrain.py`. It has a good pair of start-goal values and boundaries for the size of the big world in the front of the room. It will only work on the robot. If you want to test in simulation, you can switch back to using `mapAndRaceBrain.py`.

Checkoff 5. **Wk.14.2.6**: Post your best score for the real robot race on the blackboard, and also in Tutor problem **Wk.14.2.6**. Special prizes to the winners!